日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Dependent Polymorphism

(Preliminary Report)

Makoto Hamana

We report that an algebraic semantics of dependent types induces a kind of polymorphism, which we call dependent polymorphism. This might be a principle of dependently-typed programming.

1 Introduction

Dependently-typed programming has already been available to all programmers. Agda [8], Epigram [6], Coq with PROGRAM tactic [11], and contemporary Haskell with GADTs [10] have offered a variety of new programming techniques using dependent types [9]. But it has not been clear what good principles of dependently-type programming are.

In this note, we show a principle of dependently-type programming derived from semantics. We firstly give an algebraic semantics of inductive families based on Fiore's semantics of dependently-sorted abstract syntax [2]. Then we show that it gives a kind of polymorphism on programs, which we call dependent polymorphism. This might be a useful principle of dependently-typed programming.

2 Another Semantics of Inductive Families

Inductive families are a principal feature of dependently-typed programming. They are an indexed version of inductive datatypes.

The most basic inductive families are the usual inductive datatypes. For example, the following is the definition of inductive type of natural numbers using Agda's notation.

$$\begin{array}{l}\textbf{data}\ \mathit{Nat} : \mathit{Set}\ \textbf{where}\\ \quad \mathit{zero} : \mathit{Nat}\\ \quad \mathit{suc} : \mathit{Nat} \to \mathit{Nat}\end{array}$$

This notation is also the same as that for GADTs (Generalised Algebraic Datatypes) in Haskell.

Truly inductive families are given by indexed types. For example, we can define an inductive family *Vec* of length-indexed lists (called *vectors*) as a type indexed by *Nat*:

$$\begin{array}{l}\textbf{data}\ \mathit{Vec} : \mathit{Nat} \to \mathit{Set}\ \textbf{where}\\ \quad \mathit{nil} : \mathit{Vec}\ \mathit{Zero}\\ \quad \mathit{cons} : (n : \mathit{Nat}) \times (b : B) \times \mathit{Vec}\ n \to \mathit{Vec}\ (\mathit{suc}\ n)\end{array}$$

Initial algebra semantics of inductive families, such as *Nat* and *Vec*, has been known by dependent polynomial functors [7] [3] or indexed functors [1].

In this section, we give yet another algebraic semantics based on Fiore's algebraic semantics of dependently-sorted abstract syntax, which is now tailored for inductive families. Fiore's semantics is different from any other existing approach, and is more natural semantics of inductive families than existing ones from the viewpoint of dependently-typed programming.

2.1 Sorts

Let $\mathbb{S}$ be a directed acyclic graph of sorts expressing dependency between sorts. Then it generates a free category $\hat{\mathbb{S}}$, called *dependency category*. We

Dependent Polymorphism
Makoto Hamana, 群馬大学 工学研究科, Department of Computer Science, Gunma University.

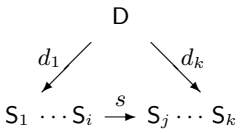
abuse to denote $\hat{\mathbb{S}}$ as $\mathbb{S}$. Note that $\mathbb{S}$ forms a simple category [5]. A sort in $\mathbb{S}$ is typically denoted by $\mathsf{S}, \mathsf{S}_1, \ldots, \mathsf{D}$, etc. We call a map $d : \mathsf{S} \to \mathsf{S}'$ in $\mathbb{S}$ *(sort) dependency*.

Each sort $\mathsf{D} \in \mathbb{S}$ is used for the name of an inductive family, so we also call D an inductive family.

A *specification of an inductive family* $\mathsf{D} \in \mathbb{S}$ is of the form

$$\mathsf{D} : (x_1 : S_1, \ldots, x_k : S_k) \to \mathrm{sort}$$

where $(x_1 : S_1, \ldots, x_k : S_k)$ is a *telescope* (or, *context*) – a sequence of declarations $x_i : S_i$ where later types may depend on earlier variables. The specification must be consistent with sort dependency in $\mathbb{S}$, i.e. for each S_i, there must uniquely exist a dependency $d : \mathsf{D} \to \mathsf{S}_i$ in $\mathbb{S}$. This is read as "D *depends on* S_i".



2. 2 Signatures

A *signature* Σ is a finite set of specifications of constructors. A *specification of constructor* of an inductive family D is of the form

$$c : (x_1 : S_1, \ldots, x_k : S_k) \to \mathsf{D}(t_1, \ldots, t_l)$$

where each t_i is a well-formed term under the context $x_1 : S_1, \ldots, x_k : S_k$. For that specification, we may call $(x_1 : S_1, \ldots, x_k : S_k)$ *source sorts*, and $\mathsf{D}(t_1, \ldots, t_l)$ a *target sort*.

Convension 2.1 We may call a sort-with-parameters $\mathsf{D}(t_1, \ldots, t_l)$ simply a sort. We typically use letters $S, S_1, \ldots, D$ in italic to range over sort-with-parameters, and letters $\mathsf{S}, \mathsf{S}_1, \ldots, \mathsf{D}$ in san-selif to range over the corresponding sorts.

2. 3 Terms

Well-formed terms are derived by the following rules of term judgments:

$$\frac{x : S \in \Delta}{\Delta \vdash x : S}$$

$$\frac{c : (x_1 : S_1, \ldots, x_k : S_k) \to D \in \Sigma \qquad \begin{array}{l}\Delta \vdash t_1 : S_1 \\ \cdots \\ \Delta \vdash t_n : S_n\end{array}}{\Delta \vdash c(t_1, \ldots, t_n) : D}$$

Here Δ is a telescope.

2. 4 Sort Definitions

Any signature Σ for $\mathbb{S}$ is partitioned into $|\mathbb{S}|$-parts, where each partition defines an inductive family D in Σ. That is, a partition for a sort D is a set of specifications having the same target sort D (with possibly different parameters):

$$c_1 : \Delta_1 \to \mathsf{D}(t_1^1, \ldots, t_l^1)$$
$$\cdots$$
$$c_n : \Delta_n \to \mathsf{D}(t_1^n, \ldots, t_l^n)$$

We denote this partition for the sort D by Σ^{D} and call *the signature for* D.

Given a signature Σ, we can construct a sequence of partitions of signature

$$\Sigma^{\mathsf{D}_1},\ \Sigma^{\mathsf{D}_2},\ \ldots\ \Sigma^{\mathsf{D}_n}$$

that satisfies the following ordering condition: when there exists $d : \mathsf{D}_i \leftarrow \mathsf{D}_j$ in S, always $i < j$. This means that a later signature depends on an earlier signature. We call the sequence of sorts appearing in the above signatures

$$\mathsf{D}_1, \ldots, \mathsf{D}_n$$

a *sort dependency sequence*. Now, a later sort depends on an earlier sort. For a sort D_i in a sort dependency sequence, the sort D_{i-1} is called the *predecessor* of D_i.

2. 5 Overview

Let D_1, D_2, $\ldots$ D_n be a sort dependency sequence. Our strategy to define the notion of models for sorts is by using course-of-value induction on n.

A D_i*-model* A is a presheaf in $\mathbf{Set}^{\mathbb{S}}$ equipped with the interpretation c^A for each constructor c of the target sort D_i. We will later define the structure of D_i-model more precisely with

- the category D_i**-Mod** of D_i-models

equipped with

- a forgetful functor

$$\underline{(-)} : \mathsf{D}_i\textbf{-Mod} \longrightarrow \mathsf{D}_{i-1}\textbf{-Mod}.$$

for $i = 2, \ldots, n$.

The base case is for the models of the most basic sort D_1. By definition of dependency category $\mathbb{S}$, the sort D_1 must be non-parametric, i.e. the specification of D_1 is

$$\mathsf{D}_1 : () \to \mathrm{sort}$$

and its signature Σ^{D_1} is of the form

$$c_1 : \Delta_1 \to \mathsf{D}_1 \quad \cdots \quad c_n : \Delta_n \to \mathsf{D}_1$$

where each Δ_i is of the form $(x_1 : \mathsf{D}_1, \ldots, x_k : \mathsf{D}_1)$, i.e. every target is always the sort D_1. This means that the sort D_1 must be the usual inductive type. The signature Σ^{D_1} gives rise the signature functor $\Sigma^{\mathsf{D}_1} : \mathbf{Set}^{\mathbb{S}} \to \mathbf{Set}^{\mathbb{S}}$ by

$$(\Sigma^{\mathsf{D}_1} A)_{D_1} = \coprod_{i=1,\ldots,n} A_{D_1}^{|\Delta_i|}$$
$$(\Sigma^{\mathsf{D}_1} A)_S = 0 \qquad (S \neq D_1)$$

A Σ^{D_1}-algebra is a pair $(A, \alpha : \Sigma^{\mathsf{D}_1} A \to A)$. A D_1-model is defined to be a Σ^{D_1}-algebra. This gives the meaning of each constructor of the target sort D_1. Then we define the category D_1**-Mod** as the category of all D_1-models and homomorphisms.

2.6 The category of models

Suppose $\mathsf{D}_k, \mathsf{D}_{k-1}, \ldots, \mathsf{D}_1, \mathsf{D}$ to be a dependency sort sequence. In general, for a D-model A, we get the corresponding models by applying the forgetful functor $(-)$:

$$\underline{A} \in \mathsf{D}_1\textbf{-Mod}, \quad \underline{\underline{A}} \in \mathsf{D}_2\textbf{-Mod}, \quad \ldots$$
$$\underline{(A)}_k \in \mathsf{D}_k\textbf{-Mod}.$$

where the last one denotes the k-times application of the forgetful functor.

This means that when constructing a D-model A, we can always track any previously constructed model that D depends on. We write $\underline{(A)}_i$ as $\underline{(A)}_{\mathsf{D}_i}$. Each $\underline{(A)}_{\mathsf{D}_i} \in \mathsf{D}_i$**-Mod** is basically for giving the meaning of $c : \Delta \to \mathsf{D}_i$.

2.7 Interpretation of Source Sorts and Terms

Given a well-typed term $\Delta \vdash \tau : S$, its interpretation in a S-model A is a function

$$[\![\Delta \vdash \tau : S]\!]_A : [\![\Delta]\!]_A \to A_{\mathsf{S}}.$$

The interpretation of term judgments is defined by

$$[\![\Delta \vdash x : S]\!]_A = \pi_x$$
$$[\![\Delta \vdash c(\tau_1, \ldots, \tau_n) : S]\!]_A =$$
$$c^A \circ \langle [\![\Delta \vdash \tau_1 : S_1]\!]_{\underline{(A)}_{\mathsf{S}_1}}, \ldots, [\![\Delta \vdash \tau_n : S_n]\!]_{\underline{(A)}_{\mathsf{S}_n}} \rangle$$

where c^A is the operation of S in the S-model A.

Given a telescope $\Delta = (x_1 : S_1, \ldots, x_n : S_n)$, its interpretation in a S-model A is a set

$$[\![\Delta]\!]_A = \{(a_1, \ldots, a_n) \in A_{\mathsf{S}_1} \times \cdots \times A_{\mathsf{S}_n} \mid$$
$$S_i = \mathsf{S}_i(\ldots, \tau, \ldots),\ \tau : S,$$
$$2 \leq i \leq n, \quad d : \mathsf{S}_i \to \mathsf{S},$$
$$A_{\mathsf{S}} \ni A_d(a_i) = [\![\Delta_{i-1} \vdash \tau : S]\!]_A(a_1, \ldots, a_{i-1})\}$$

where $\Delta_{i-1} = (x_1 : S_1, \ldots, x_{i-1} : S_{i-1})$.

Note that intuitively, $a_i : \mathsf{S}_i(\ldots, \tau, \ldots)$.

2.8 Signature Functor

Let D be a sort and D' its predecessor. Let Σ^{D} and $\Sigma^{\mathsf{D}'}$ be signatures. We define the signature functor for the sort D

$$\Sigma^{\mathsf{D}} : \mathsf{D}'\textbf{-Mod} \longrightarrow \mathbf{Set}^{\mathbb{S}},$$

which takes a D'-model $\underline{A}$ (with underlying presheaf $A \in \mathbf{Set}^{\mathbb{S}}$) and a sort in $\mathbb{S}$ and returns a set as follows:

$$(\Sigma^{\mathsf{D}} \underline{A})_{\mathsf{D}} = [\![\Delta_1]\!]_{\underline{A}} + \cdots + [\![\Delta_n]\!]_{\underline{A}}$$
$$(\Sigma^{\mathsf{D}} \underline{A})_{d_i} = [\,[\![\Delta_1 \vdash t_i^1 : S_i]\!]_{\underline{A}}, \ldots, [\![\Delta_n \vdash t_i^n : S_i]\!]_{\underline{A}}\,] : (\Sigma^{\mathsf{D}} \underline{A})_{\mathsf{D}} \to (\Sigma^{\mathsf{D}} \underline{A})_{\mathsf{S}_i}$$
$$(\Sigma^{\mathsf{D}} \underline{A})_{\mathrm{id}_{\mathsf{D}}} = \mathrm{id}_{[\![\Delta_1]\!]_{\underline{A}} + \cdots + [\![\Delta_n]\!]_{\underline{A}}}$$
$$(\Sigma^{\mathsf{D}} \underline{A})_{\mathsf{S}_i} = A_{\mathsf{S}_i}$$
$$(\Sigma^{\mathsf{D}} \underline{A})_{\mathrm{id}_{\mathsf{S}_i}} = \mathrm{id}_{A_{\mathsf{S}_i}}$$
$$(\Sigma^{\mathsf{D}} \underline{A})_s = A_s$$

where

- $d_i : \mathsf{D} \to \mathsf{S}_i,\ s : \mathsf{S}_i \to \mathsf{S}_j$ in $\mathbb{S}$ are non-identities $(0 \leq i \leq l)$,
- D has the specification

$$c_1 : \Delta_1 \to \mathsf{D}(t_1^1, \ldots, t_l^1)$$
$$\ldots$$
$$c_n : \Delta_n \to \mathsf{D}(t_1^n, \ldots, t_l^n).$$

If $\mathsf{S} \in \mathbb{S}$ is unreachable from D, we set

$$(\Sigma A)_{\mathsf{S}} = 0$$
$$(\Sigma A)_d = ! : 0 \to A_{\mathsf{S}'}$$
$$(\Sigma A)_{\mathrm{id}_{\mathsf{S}}} = \mathrm{id}_0$$

where $d : \mathsf{S} \to \mathsf{S}'$ in $\mathbb{S}$ is a non-identity.

2.9 Algebras and Models

Let D be a sort and D' its predecessor. A Σ^{D}-*algebra* is a pair $(\underline{A}, \alpha)$:

- a D'-model $\underline{A}$ with
- underlying presheaf $A \in \mathbf{Set}^{\mathbb{S}}$, and
- a map $\alpha_D = [c^A]_{c \in \Sigma^{\mathsf{D}}} : (\Sigma^{\mathsf{D}} \underline{A})_D \to A_D$ in

$\mathbf{Set}^{\mathbb{S}}$, called *algebra structure*, where c^A is an *operation*

$$c^A : [\![\Delta]\!]_A \to A_{\mathsf{D}}$$

defined for each $c : \Delta \to D \in \Sigma^{\mathsf{D}}$.

A D-*model* is a Σ^{D}-algebra $A = (\underline{A}, \alpha)$ such that

$$\alpha_{\mathsf{S}} = \mathrm{id}_{A_{\mathsf{S}}}$$

for all sort S ($\neq \mathsf{D}$) which is reachable from D.

Obviously, Σ^{D}-algebras and homomorphisms, D-models and homomorphisms form categories Σ^{D} **-Alg**, D **-Mod**, respectively.

3 Dependent Polymorphism

Using the semantics in the previous section, we observe an interesting phenomenon on a dependently-typed function.

We consider the inductive families *Nat* and *Vec* defined in Introduction. We suppose two sorts N and V corresponding to *Nat* and *Vec*. Since *Vec* depends on *Nat*, we set the dependency category $\mathbb{S}$ as

$$\boxed{\mathsf{N} \xleftarrow{\mathsf{len}} \mathsf{V}}$$

where len is the sort dependency and identities are ommited. This is read as that V depends on N. Notice that the name len of the sort dependency is not important, and len is merely an arrow (not a function).

In the category $\mathbf{Set}^{\mathbb{S}}$, we can model *Vec* and *Nat*. We consider a term model $T \in \mathbf{Set}^{\mathbb{S}}$:

$$T_{\mathsf{N}} = \{zero\} \cup \{suc(n) \mid n \in T_{\mathsf{N}}\}$$

$$T_{\mathsf{V}} = \{nil\} \cup \{cons(n, b, y) \mid n \in T_{\mathsf{N}}, b \in T_B, y \in T_{\mathsf{V}}, T(\mathsf{len})(y) = n\}$$

which T gives a V-model.

By functoriality, $T : \mathbb{S} \to \mathbf{Set}$ maps the sort dependency len to a function $T(\mathsf{len})$

$$\begin{array}{ccc} \mathsf{N} & \xleftarrow{\mathsf{len}} & \mathsf{V} \quad \text{in } \mathbb{S} \\ \downarrow{\scriptstyle T} & & \\ T_{\mathsf{N}} & \xleftarrow[T(\mathsf{len})]{} & T_{\mathsf{V}} \quad \text{in } \mathbf{Set} \end{array}$$

defined by

$$T(\mathsf{len})(nil) = 0$$

$$T(\mathsf{len})(cons(n, a, y)) = suc(n)$$

i.e., the length function on vectors. This functoriality is automatically imposed by the conditions of semantics described in the previous section.

Consider the standard append function $+\!\!+$ on vectors and the addition on natural numbers.

$$\_ +\!\!+ \_ : Vec(m) \times Vec(n) \to Vec(m+n)$$

$$nil +\!\!+ ys = ys$$

$$(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)$$

$$\_ + \_ : Nat \to Nat$$

$$zero + y = y$$

$$suc(n) + y = suc(n + y)$$

Then the following diagram obviously commutes.

$$\begin{array}{ccc} \mathsf{V} & & T_{\mathsf{V}} \times T_{\mathsf{V}} \xrightarrow{+\!\!+} T_{\mathsf{V}} \\ \downarrow{\scriptstyle \mathsf{len}} & & \downarrow{\scriptstyle T\mathsf{len} \times T\mathsf{len}} \qquad \downarrow{\scriptstyle T\mathsf{len}} \\ \mathsf{N} & & T_{\mathsf{N}} \times T_{\mathsf{N}} \xrightarrow[+]{} T_{\mathsf{N}} \end{array}$$

in $\mathbb{S}$ in **Set**

One way of understanding this diagram is to regard it as the naturality square of a certain natural transformation $\oplus : T \to T$.

How can we define "$\oplus$"? A candidate is to schematically define it as

$$z \oplus y = y$$

$$c(n) \oplus y = c(n \oplus y)$$

This means that $\oplus$ is instantiated to $+\!\!+$ at the component V by

- interpreting z as nil, and
- interpreting c as $(x :)$,

and is instantiated to $+$ at the component N by

- interpreting z as $zero$, and
- interpreting c as suc.

Then the above naturality square is seen as expressing *polymorphism* of the "schematic" function $\oplus$. This polymorphism is restricted to the dependency len, hence we call it dependent polymorphism.

This phenomena of dependent polymorphism is typical for an inductive family indexed by some "shape" of datatype. In the above case, natural numbers are seen as the shapes of vectors. There are also many other examples of such shapely-indexed data types. This might be relevant to Barry Jay's shapely types and polymorphism [4].

Acknowledgement. I am grateful to Marcelo Fiore for fruitful discussions on the semantics of dependently-sorted abstract syntax.

参 考 文 献

[1] Altenkirch, T. and Morris, P.: Indexed Containers, *LICS*, 2009, pp. 277–285.

[2] Fiore, M. P.: Second-Order and Dependently-Sorted Abstract Syntax, *LICS*, 2008, pp. 57–68.

[3] Gambino, N. and Hyland, M.: Wellfounded Trees and Dependent Polynomial Functors, *TYPES*, 2003, pp. 210–225.

[4] Jay, C. B. and Cockett, J.: Shapely Types and Shape Polymorphism, *ESOP*, 1994, pp. 302–316.

[5] Makkai, M.: First-order logic with dependent sorts, with applications to category theory, 1997. Preprint.

[6] McBride, C. and McKinna, J.: The view from the left, *Journal of Functional Programming*, Vol. 14, No. 1(2004), pp. 69–111.

[7] Moerdijk, I. and Palmgren, E.: Wellfounded trees in categories, *Annals of Pure and Applied Logic*, Vol. 104(2000), pp. 189–218.

[8] Norell, U.: *Towards a practical programming language based on dependent type theory*, PhD Thesis, Chalmers University of Technology, 2007.

[9] Oury, N. and Swierstra, W.: The power of Pi, *Proc. of ICFP'08*, 2008, pp. 39–50.

[10] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G.: Simple unification-based type inference for GADTs, *Proc. of ICFP '06*, 2006, pp. 50–61.

[11] Sozeau, M.: Program-ing Finger Trees in Coq, *ICFP'07*, 2007, pp. 13–24.